什么是动态规划?

动态规划（英语：Dynamic programming，简称 DP），是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有重叠子问题和最优子结构性质的问题
核心思想: 通过将问题拆分成一个一个小问题，记录过往结果，减少重复运算
小例子:

```
A ： "1+1+1+1+1+1+1+1 =？"
A ： "上面等式的值是多少"
B ： 计算 "8"
A ： 在上面等式的左边写上 "1+" 呢？
A ： "此时等式的值为多少"
B ： 很快得出答案 "9"
A ： "你怎么这么快就知道答案了"
A ： "只要在8的基础上加1就行了"
A ： "所以你不用重新计算，因为你记住了第一个等式的值为8!动态规划算法也可以说是 '记住求过的解来节省时间'"
```

什么是背包问题?

背包问题(Knapsack problem)是一种组合优化的NP完全问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。问题的名称来源于如何选择最合适的物品放置于给定背包中。相似问题经常出现在商业、组合数学，计算复杂性理论、密码学和应用数学等领域中。也可以将背包问题描述为决定性问题，即在总重量不超过W的前提下，总价值是否能达到V？它是在1978年由Merkle和Hellman提出的。

01背包

背包 DP

前置知识：动态规划部分简介。

引入

在具体讲何为「背包 dp」前，先来看如下的例题：

> 「USACO07 DEC」Charm Bracelet
>
> 题意概要：有 n 个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

在上述例题中，由于每个物体只有两种可能的状态（取与不取），对应二进制中的 0 和 1，这类问题便被称为「0-1 背包问题」。

例题中已知条件有第i个物品的重量 w_i，价值 v_i，以及背包的总容量 W。

设 DP 状态 $f_{i,j}$为在只能放前i个物品的情况下，容量为j的背包所能达到的最大总价值。
那么对于容量为j，最大总价值为$f_{i,j}$，前i个物品一定有一种选取方式成立
考虑转移。假设当前已经处理好了前$i-1$个物品的所有状态，那么对于第i个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为$f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小，背包中物品的总价值会增大，故这种情况的最大价值为$f_{i-1,j-w_i}+v_i$。

由此可以得出状态转移方程：$$f_{i,j}=\max(f_{i-1,j},f_{i-1,j-w_i}+v_i)$$

实现：

代码块1
```
// C++ Version
for(int i = 1; i <= n; i++){
    for(int j = 0; j <= W - w[i]; j++){
        f[i][j + w[i]] = max(f[i - 1][j] + v[i],f[i - 1][j + w[i]]);
    }
    for(int j = 0; j <= w[i] - 1; j++) f[i][j] = f[i - 1][j];
}
```

上述代码的空间复杂度和时间复杂度都是 O(nw)
有什么办法可以降低空间复杂度吗?
我可以发现每一个$f_{i,j}$他只与$f_{i-1,j}$有关，所以当我们遍历到i的时候$i-1$以前的状态我们都可以舍弃不要，所以我们每次只需要记录两种状态就可以了，所以我们可以采用滚动数组。

代码块2
```
// C++ Version
for(int i = 1; i <= n; i++){
    for(int j = 0; j <= w[i] - 1; j++) f[i % 2][j] = f[(i - 1) % 2][j];
    for(int j = 0; j <= W - w[i]; j++){
        f[i % 2][j + w[i]] = max(f[(i - 1) % 2][j + w[i]],f[(i - 1) % 2][j] +
v[i]);
    }
}
```

那么能不能再简化呢?
如果我们只用一维行不行?

代码块3
```
// C++ Version
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++)
    f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$，在 $j\geqslant w_i$ 时，$f_{i,j}$ 是会被 $f_{i,j-w_i}$ 所影响的。这就相当于物品 i 可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i，这样就不会出现上述的错误，因为 $f_{i,j}$ 总是在 $f_{i,j-w_i}$ 前被更新。

```
// C++ Version
for (int i = 1; i <= n; i++)
  for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
//需要对0~w[i] - 1的进行操作吗?
```

为什么这样枚举就是正确的呢?
大家可以自己的想一想，我们每次更新的f_l是从大到小更新的，而没有更新过的都是上一物品转换完的状态。

完全背包

> 完全背包
>
> 解释
>
> 完全背包模型与 0-1 背包类似，与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。
>
> 我们可以借鉴 0-1 背包的思路，进行状态定义：设 $f_{i,j}$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。
>
> 需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同。

过程

可以考虑一个朴素的做法：对于第 i 件物品，枚举其选了多少个来转移。这样做的时间复杂度是 $O(n^3)$ 的。

状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{+\infty}(f_{i-1,j-k\times w_i} + v_i \times k)$$

这样的枚举方法就是把完全背包转换成了01背包，把一个物品i当作一个物品，把两个物品i当作一个物品，...，把n个物品i当作一个物品。

代码块4
```
// C++ Version
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++){
      for(int k = 0;;k++){
          if(k * w[i] + l > W) break;
          f[i][l + k * w[i]] = max(f[i][l + k * w[i]],max(f[i - 1][l + k *
w[i]],f[i - 1][l] + k * v[i]));
      }
      for(int l = 0; l <= w[i] - 1; l++) f[i][l] = f[i - 1][l];
  }
```

那么我们如何进行一个优化呢？我们换个思想去想，只要我们给物品i足够的个数，再取01背包，这样看来也是完全背包。
物品1可以取无限次，和有无限个物品1，每个只能取一次，这俩个是等价的

考虑做一个简单的优化。可以发现，对于 $f_{i,j}$，只要通过 $f_{i,j-w_i}$ 转移就可以了。因此状态转移方程为：

$$f_{i,j} = \max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

代码块5

```
// C++ Version
for (int i = 1; i <= n; i++){
  for(int l = 0; l <= w; l++) f[i][l] = f[i - 1][l];
  for (int l = 0; l <= W - w[i]; l++)
    f[i][l + w[i]] = max(f[i][l + w[i]],f[i][l] + v[i]));
}
```

随后我们发现$f_{i,j}$只和自己这些物品有关系

代码块6

```
// C++ Version
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++)
    f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
```

多重背包

多重背包

多重背包也是 0-1 背包的一个变式。与 0-1 背包的区别在于每种物品有 k_i 个，而非一个。

一个很朴素的想法就是：把「每种物品选 k_i 次」等价转换为「有 k_i 个相同的物品，每个物品选一次」。这样就转换成了一个 0-1 背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{k_i}(f_{i-1,j-k\times w_i} + v_i \times k)$$

时间复杂度 $O(W\sum_{i=1}^{n} k_i)$。

```
for(int i = 1; i <= n; i++){
    for(int j = 0; j <= W; j++){
        for(int l = 0; l <= k[i]; l++){
            if(j + l * c[i] > w) break;
            dp[i % 2][j + l * c[i]] = max(dp[i % 2][j + l * c[i]],dp[(i - 1) %
2][j] + v[i]);
        }
    }
}
```

二进制分组优化 ¶

考虑优化。我们仍考虑把多重背包转化成 0-1 背包模型来求解。

解释

显然，复杂度中的 $O(nW)$ 部分无法再优化了，我们只能从 $O(\sum k_i)$ 处入手。为了表述方便，我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。

在朴素的做法中，$\forall j \leq k_i$，$A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。举例来说，我们考虑了「同时选 $A_{i,1}, A_{i,2}$」与「同时选 $A_{i,2}, A_{i,3}$」这两个完全等效的情况。这样的重复性工作我们进行了许多次。那么优化拆分方式就成为了解决问题的突破口。

过程

我们可以通过「二进制分组」的方式使拆分方式更加优美。

具体地说就是令 $A_{i,j}\ (j \in [0, \lfloor \log_2(k_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。特殊地，若 k_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $k_i - 2^{\lfloor \log_2(k_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$
- $18 = 1 + 2 + 4 + 8 + 3$
- $31 = 1 + 2 + 4 + 8 + 16$

显然，通过上述拆分方式，可以表示任意 $\leq k_i$ 个物品的等效选择方式。将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可。

时间复杂度 $O(W \sum_{i=1}^{n} \log_2 k_i)$

典型的空间复杂度换取时间复杂度的做法

```
// C++ Version
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k - c > 0) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}
```

混合背包

混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取 k 次。

这种题目看起来很吓人，可是只要领悟了前面几种背包的中心思想，并将其合并在一起就可以了。下面给出伪代码：

```
for (循环物品种类) {
  if (是 0 - 1 背包)
    套用 0 - 1 背包代码;
  else if (是完全背包)
    套用完全背包代码;
  else if (是多重背包)
    套用多重背包代码;
}
```

二维费用背包

二维费用背包

> 「Luogu P1855」榨取 kkksc03
>
> 有 n 个任务需要完成，完成第 i 个任务需要花费 t_i 分钟，产生 c_i 元的开支。
>
> 现在有 T 分钟时间，W 元钱来处理这些任务，求最多能完成多少任务。

这道题是很明显的 0-1 背包问题，可是不同的是选一个物品会消耗两种价值（经费、时间），只需在状态中增加一维存放第二种价值即可。

这时候就要注意，再开一维存放物品编号就不合适了，因为容易 MLE。

```
f[i][j][k]表示前i个任务花费了j分支k元产生的最大价值
f[i][j][k] = max(f[i][j][k],f[i][j - t[i]][k - c[i]] + w[i]);
```

分组背包

分组背包

> 「Luogu P1757」通天之分组背包
>
> 有 n 件物品和一个大小为 m 的背包，第 i 个物品的价值为 w_i，体积为 v_i。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了。

再说一说如何进行存储。我们可以将 $t_{k,i}$ 表示第 k 组的第 i 件物品的编号是多少，再用 cnt_k 表示第 k 组物品有多少个。

```
// C++ Version
for (int k = 1; k <= ts; k++)          // 循环每一组
  for (int i = m; i >= 0; i--)         // 循环背包容量
    for (int j = 1; j <= cnt[k]; j++)  // 循环该组的每一个物品
      if (i >= w[t[k][j]])
        dp[i] = max(dp[i],dp[i - w[t[k][j]]] + c[t[k][j]]);  // 像0-1背包一样状态转移
```

有依赖的背包

有依赖的背包

> 「Luogu P1064」金明的预算方案
>
> 金明有 n 元钱，想要买 m 个物品，第 i 件物品的价格为 v_i，重要度为 p_i。有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件。
>
> 目标是让所有购买的物品的 $v_i \times p_i$ 之和最大。

考虑分类讨论。对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件 + 某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。

如果是多叉树的集合，则要先算子节点的集合，最后算父节点的集合。

当作分组背包来看，主件，附件 + 主件，只能买一样

其他：

杂项

小优化

根据贪心原理，当费用相同时，只需保留价值最高的；当价值一定时，只需保留费用最低的；当有两件物品 i, j 且 i 的价值大于 j 的价值并且 i 的费用小于 j 的费用是，只需保留 j。

背包问题变种

实现

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的。我们可以用 $g_{i,v}$ 表示第 i 件物品占用空间为 v 的时候是否选择了此物品。然后在转移时记录是选用了哪一种策略（选或不选）。
输出时的伪代码：

```
int v = V;  // 记录当前的存储空间
// 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
for (从最后一件循环至第一件) {
  if (g[i][v]) {
    选了第 i 项物品;
    v -= 第 i 项物品的重量;
  } else {
    未选第 i 项物品;
  }
}
//大家对于输出方案只需要思考，如果我们现在的最大空间是W,最大的价值是V
//一定是有一种方案使得价值是最大
//最后一点空间一定是由我们最后买的那件商品去填补的
//所以如果最后一件商品在方案中 一定存在 dp[W] == dp[W - w[i]] + v[i];
```

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数。

这种问题就是把求最大值换成求和即可。

例如 0-1 背包问题的转移方程就变成了：

$$dp_i = \sum(dp_i, dp_{i-c_i})$$

初始条件：$dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装。